DESIGN AND SIMULATION OF PERL RISC

by

T. S. BALAJI

SE
97
1
AL
ES

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY 1997

DESIGN AND SIMULATION OF PERL RISC

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
T. S. BALAJI

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
January, 1997

CSE-1997-M-BAL-DES

Certificate

Certified that the work contained in the thesis entitled "*DESIGN AND SIMULATION OF PERL RISC*", by Mr.*T. S. BALAJI*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

(Dr. Rajat Moona)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

January, 1997

Abstract

The objective of this thesis is to present **PERL RISC**, a register-less RISC architecture. This architecture embraces all the properties of RISC machines and tries to remove the bottleneck found in current day RISC machines. It is designed to be a memory-to-memory architecture because of its relevance today. The availability of large on-chip caches and wider bus bandwidth to reduce memory latency supports our idea. The instruction set and pipeline design of this architecture are presented. The thesis also explains the design and implementation of **SuperSIM**, an instruction set simulator for **PERL RISC**. This superscalar simulator achieves the most aggressive issue policy: multiple out-of-order issue and multiple out-of-order execution. Some powerful hardware mechanisms used to achieve this policy, like a central instruction window, reorder buffers etc., are explained in detail. To further improve performance, the simulator performs branch prediction to dynamically schedule instructions that follow a conditional branch. Finally, the performance of some typical user programs on this machine are compared with similar results obtained for current day RISC machines.

Acknowledgments

I would like to thank **Dr. Rajat Moona** whose inquisitiveness and enthusiasm prompted me to delve deep into the exotic world of microprocessors and understand their architecture better. I also wish to thank Suresh and Ramana Kumar for their cooperation in this project. I would also like to express my gratitude to all the programmers in the computing community who make their work available to others which helps in developing better quality software and makes computing a pleasure. I would also like to thank all my family members who supported me during my entire stay at Kanpur. And last, but certainly not the least, I would also like to thank each and every one of M.Tech., 95 batch for making my stay at IIT-K a memorable one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

PERL RISC (Performance Enhanced Register-Less RISC) architecture is a new generation RISC architecture which we are proposing based on our understanding of various architectures proposed and implemented in the past. This new architecture embraces fundamental concepts of simplicity and general applicability while extending itself to the advanced techniques that will carry it into the next decade and beyond.

SuperSIM, the simulator for a superscalar version of **PERL RISC** architecture, is implemented for analyzing this new architecture. This simulator is capable of executing assembly language programs of **PERL RISC** architecture and provide us with statistics which are compared with implementations of other popular and state-of-the-art RISC architectures.

1.1 Motivation

There are many factors which influenced us in the design of this processor. Firstly, current day microprocessors depend extensively on on-chip cache to offset the memory latencies. Secondly, current growth in technology helps in building caches which are cheaper and comparable in speed to registers. Thirdly, the Load/Store overhead associated with all the RISC machines of today. All these are discussed in detail in the following sections. However we followed the RISC philosophy because of the obvious advantages it offers, like simplified control unit, easy fetch and decode logic and efficient handling of pipelines due to uniformity of instruction size and duration of execution. These factors contribute significantly to the increase in computing speed.

1.2 The Origins of RISC

During the 70s, memory efficiency was an important factor which influenced architecture design because main memory (i.e., magnetic core) was slow and expensive. This resulted in much pervasive thought that larger programs are invariably slower programs. It also urged many designers to design instruction sets which had instructions that reduced program size. There were machine instructions that closely resembeled programming language statements, thereby reducing the "semantic gap" between these languages. But this introduced a "performance gap" in these machines because the decoding logic became complex and pipelining was difficult.

The new RISC philosophy of design evolved. The main idea of RISC is to have simpler instructions so as to reduce the clock cycle time. As these machines had simple and minimum number of instructions these are called *Reduced Instruction Set Computers.*

1.2.1 Common RISC Traits

Most of the RISC machines of today have the following in common:

1. All instructions have fixed size and simple format.

2. These machines have only a handful (upto 4) of addressing modes.

3. They use hardwired logic for control.

4. Only Load/Store instructions access memory.

5. All arithmetic operations are performed on registers with no instruction combining Load/Store with arithmetic.

6. They aim at more performance from current compiler technology.

1.3 RISC Variations

Each RISC machine provides its own particular variations on the common theme. This makes for some interesting differences. The following discussion highlights the variants.

1.3.1 Register Windows

RISC machines used hardwired control. As a result, more die area was available on-chip. As the memory technology at that time did not allow faster caches (also use of caches was in its rudimentary stages), these machines had many registers. To utilize registers to maximum, state of the art compiler technology was pushed to the maximum possible limit. The first step was to have enough registers to keep all the local scalar variables and all the parameters of the current procedure in registers, as procedure calls are slowed down when there are a great many registers [Pat85]. The solution was to have many sets, or windows, of registers, so that registers would not have to be saved on every procedure call and restored on every return. The Berkeley RISC machine, rather than copying the parameters from one window to another on each call, overlapped windows so that some registers are simultaneously part of two windows. By putting parameters into the overlapping registers, operands are passed automatically.

The disadvantage of register windows are that they use more chip area and slow the basic clock cycle. Although context switches rarely occur, they require that two to three times as many registers to be saved, on average.

1.3.2 Multi-Ported Memory and Registers

One method of reducing the Load/Store overhead is to have multiple ports to memory and register. This was in fact used in Berkeley RISC to reduce costs of memory accesses [Pat85]. The 801 and MIPS solve this problem with a *delayed load*, which is analogous to the delayed branch. The idea of using delayed branch is to allow RISCs to always fetch the next instruction during the execution of the current instruction. The machine-language code is suitably arranged so that the desired results are obtained.

Though the delayed load removes the bubbles normally associated with pipelined execution, the onus is on the programmers of the compiler, the optimizer, and the debugger.

1.3.3 Multiple Instructions per Word

The idea behind this is to pack two or more instructions into every word of memory, whenever possible. This improvement could potentially provide a linear increase in performance. But data dependencies prevented some combinations to be grouped together. MIPS tried this variation, by packing two instructions into every 32-bit word whenever possible. But this resulted in an increased speedup of only 10 to 15 percent [Pat85]. Increased complexity in decoding logic and greater burden on the compiler resulted in architects not using this technique in their designs after the findings by the MIPS designers.

1.4 The Symmetric Architecture Model

The major disadvantage of RISC also is the reduced instructions. Since a RISC has a small number of instructions, a number of functions, performed on CISCs by a single instruction, will need 2, 3 or more instructions on a RISC. One way to make instructions more powerful than RISC instructions is to go one step back from the RISC's *Load/Store* architecture and use a *Symmetric* instruction set which can perform direct operation on memory [AAD90]. In a Load/Store instruction-set architecture the only instructions that access memory are LOAD and STORE. Computational instructions can only use registers as operands. A *Symmetric* instruction-set architecture, on the other hand, treats register-operands and memory-operands symmetrically, and has the additional capability to perform direct computation on memory.

The Symmetric model attempts to speedup execution by three means:

1. Reduce the path-length (the number of instructions executed for a program) by using a more powerful instruction set.

2. Eliminate the LOAD delay by using a different pipeline structure.

3. Avoid the introduction of other delays.

[AAD90] illustrates the pipeline structure for the symmetric architecture model and provide some results useful for comparison of Symmetric architecture with Load/Store architecture citing load delay as a major bottleneck in the Load/Store architecture.

1.5 Register-less RISC architecture

In Figure 1, the statement A $\leftarrow$ B+C is translated into Assembly Language for three execution models: Register-to-Register, Memory-to-Register, and Memory-to-Memory. The three data words are 32 bits each and the address field is 16 bits. Metrics were selected by research architects for deciding which architecture is best; they selected the total size of executed instructions(I), the total size of executed data (D), and the total memory traffic—that is, the sum of I and D, which is (M). These metrics suggest that a memory-to-memory architecture is the "best" architecture, and a register-to-register architecture the "worst". This study led one research architect in 1978 to suggest that "One's eyebrows should rise whenever a future architecture is developed with a register-oriented instruction set."[Mey78]

With all the above results in mind, we set about designing **PERL RISC**. To summarize, on many register machines 25 percent or more instructions are Load/Stores. The compiler should decide where to insert data movement instructions to remove load stalls. In these machines, when handling calls to subroutines, switching tasks or starting an I/O transfer it is necessary to save current machine state. This means copying all registers and restoring them. Effective use of registers demand compile time analysis. As register allocation is an NP-complete problem,

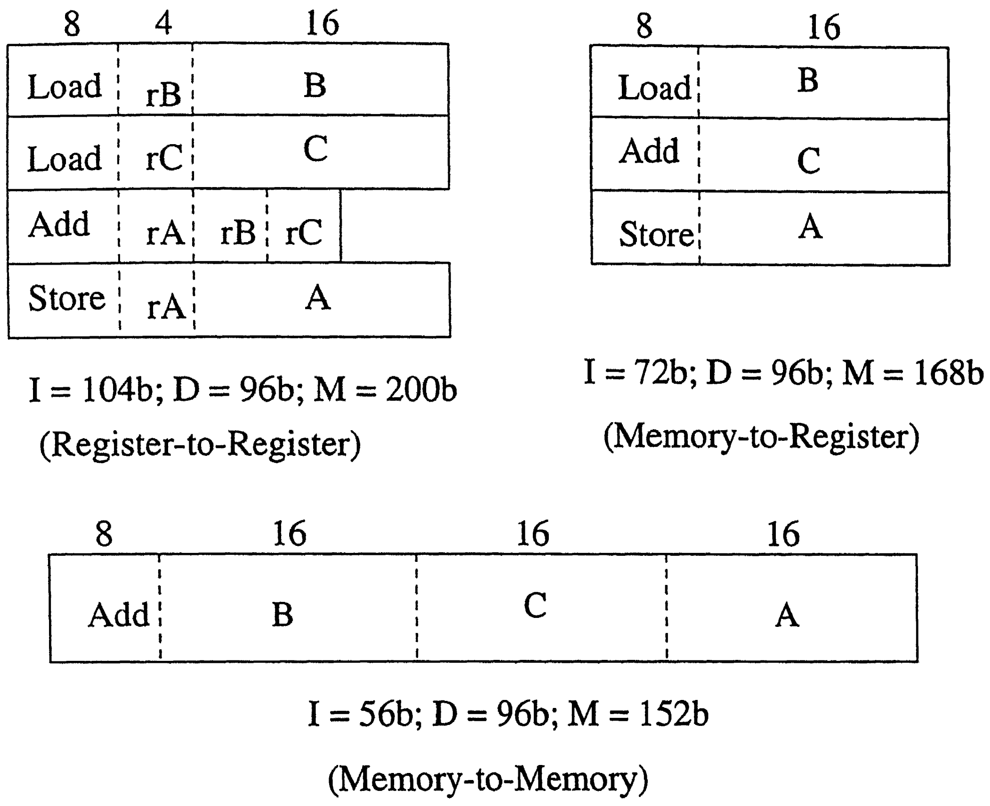

Figure 1: Instruction Set Comparison

the overhead associated with register allocation for compilers increases as the number of registers available in a machine increases. The question of optimal size of the CPU register file is a controversial one, requiring additional research. Despite some obvious advantages, certain studies have cast doubt on the benefit of a large register set [FMM87] [Wal79] [Tab94].

Sites [Sit79] discusses using Short Term Memory(STM) cells to offset the memory latency. He proposes using registers and cache for this purpose. He argues for using more STM cells to obtain better speedup and cites the speed of registers as their only advantage over caches. As increasing the number of registers beyond a certain limit (32 or 64 to the maximum) does not provide a linear performance increase, we are left with the option of using cache to offset memory latency. Also, blindly doubling the size of cache will usually have a much better performance improvement than blindly doubling the number of registers [Sit79]. As the technology today can provide caches which are as fast as registers we thought of replacing registers with caches instead of using them in tandem. The result is the birth of the new architecture.

This work is a part of the ongoing research activity in the design and performance analysis of registerless processor architecture at the Department of CSE, IIT Kanpur. This work helped in giving feedback in the design of the instruction set and the pipeline .

1.6 Organization of the Thesis

The rest of the report is organized as follows. In Chapter 2 a brief description of Instruction Level Parallelism is presented with emphasis on superscalar microprocessor design. Few case studies are also provided. Chapters 3 and 4 provide an overview of the instruction set design and pipeline design of **PERL RISC**. Chapter 5 discusses the implementation details of **SuperSIM**. Chapter 6 provides some simulation results. In Chapter 7, we highlight some extensions to our work which can be explored to understand better this nascent architecture. Appendix A lists the commands supported by **SuperSIM**. Appendix B gives an example machine description file. Appendix C gives the instruction set of **PERL RISC**. Appendix D gives an example assembly program generated by the C compiler [Kum97] for **PERL RISC** and finally Appendix E explains the statistical output generated by **SuperSIM**.

Chapter 2

Instruction Level Parallelism

Pipelining is a mechanism whereby processors overlap execution of instructions when they are independent of one another. This potential overlapping among instruction is called *Instruction Level Parallelism(ILP).* Architectures to take advantage of this kind of parallelism have been proposed. A superscalar machine is one that can issue multiple independent instructions in the same cycle. A superpipelined machine issues one instruction per cycle, but the cycle time is set much less than the typical instruction latency. A VLIW machine is like a superscalar machine, except the parallel instructions must be explicitly packed by the compiler into very long instruction words.

Superscalar processing is the latest in a long series of architectural innovations aimed at producing ever faster microprocessors. So we decided to design a superscalar version of **PERL RISC**. The following sections describe the microarchitecture of superscalar processors in general, and discuss some features of existing superscalar microprocessors.

2.1 Superscalar Processing

Introduced at the beginning of this decade, superscalar microprocessors are now being designed and produced by all the microprocessor vendors for high-end products. Although viewed by many as an extension of the RISC movement of the

1980s, superscalar implementations are in fact heading toward increasing complexity. And superscalar methods have been applied to a spectrum of instruction sets, ranging from the DEC Alpha, the "newest" RISC instruction set, to the decidedly non-RISC Intel x86 instruction set.

A typical superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcome of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analyzed for data dependencies, and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as *dynamic instruction scheduling.* Upon completion, instruction results are resequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit *ILP.*

2.2 The Microarchitecture of Superscalar processors

Figure 2 illustrates the microarchitecture of a typical superscalar processor. The major parts are instruction issue mechanism, instruction cache to supply instructions to issue mechanism, a data cache for memory operands and an interconnect that connects together various components.

2.2.1 Instruction Fetching and Branch Prediction

For a superscalar implementation to sustain the execution of multiple instructions per cycle, the fetch phase must be able to fetch multiple instructions per cycle from the cache memory. The number of instructions fetched per cycle should at least

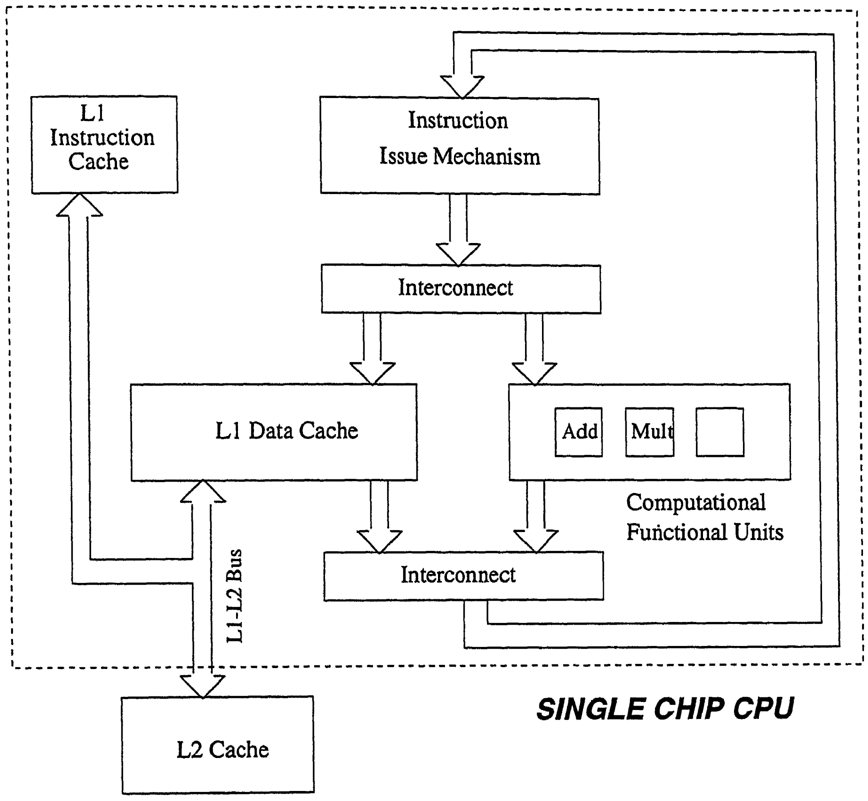

Figure 2: A Superscalar CPU

match the peak instruction decode and execution rate and is usually somewhat higher. The extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer than the maximum number of instructions can be fetched. In case of branch instructions which redirect the flow of control, however, the fetch mechanism must be redirected to fetch instructions from the branch target. Because of delays that can occur during the process of this redirection, the handling of branch instructions is critical for good performance of superscalar processors. Processing of conditional branch instructions can be broken down into the following parts:

1. recognizing that an instruction is a conditional branch. This is a trivial task of looking at the opcode and identifying the instruction type.

2. determining the branch outcome. This can be done by using some predictors. A 2 bit branch prediction scheme is currently the most widely used. Some predictors use static information i.e., information that can be determined from the static binary. Profiling information can also be used for this purpose. **SuperSIM** uses the 2 bit scheme which is discussed in detail in chapter 5.

3. computing the branch target, and

4. transferring control by redirecting instruction fetch (in the case of a taken branch).

2.2.2 Instruction Decoding and Dispatch

During this phase, instructions are removed from the instruction fetch buffers, examined, and control and data dependence linkages are set up for the renaming pipeline stages. This phase includes detection of true data dependences (due to RAW hazards) and resolution of other register hazards, e.g., WAW and WAR hazards caused by register reuse. A detailed discussion on the different classes of hazards is provided in chapter 4. Typically, this phase also distributes, or *dispatches*, instructions to buffers associated with hardware functional units for later issuing and execution. This phase works closely with the following issue stage.

2.2.3 Instruction Issue Policies

In a superscalar machine, pipeline stages are concurrent processes that may take care of a variable number of instructions per cycle. The overall coordination between processes as well as their degrees of freedom have a significant impact on the processor ability to exploit the instruction parallelism available in a program. In particular, instruction issue, which is the process of letting an instruction move from the decode stage to the execute stage determines the processor capacity to identify instructions that can be executed concurrently.

There are three policies for instruction issue as detailed below

▮ *In-order issue, in-order completion*

In this mechanism, instruction issue and commit are in exact program order. So this is the easiest mechanism to implement. Also as the saved state of the machine at any point of time is consistent with the sequential architecture model, exceptions and precise interrupts can be easily handled. But it has its own disadvantages. As parallelism is not exploited, it is very inefficient. In addition to stalling the pipeline due to data dependencies or structural hazards, all instructions that follow a long latency operation are stalled at the issue stage itself, even though there is not data dependency.

▮ *In-order issue, out-of-order completion*

In this mechanism, even though the instructions pass through the issue stage in sequential program order, they can complete execution out-of-order by bypassing each other in the execute stage. So only RAW hazards and structural hazards hinder the issue process. In particular, instruction issuing is not stalled after a long latency operation which improves performance for floating point operations. PowerPC 603 [BUOO94] is a superscalar microprocessor capable of issuing as many as 3 instructions to its 5 execution units, namely, branch, integer, floating point, Load/Store and system register. Even though instructions can execute out-of-order, and hence can complete out-of-order, they are dispatched to the 5 functional units in order.

▮ *Out-of-order issue, out-of-order completion*

Both the schemes mentioned above do not exploit parallelism to the maximum. This is because when an instruction is stalled no further instruction can proceed. But in the case of out-of-order issue, the processor looks at a window of instructions at a time. Even though stalls result in this case also due to data dependencies and structural hazards, the processor looks beyond the stalled instructions to find possible candidates for issue. This increases the processor "look-ahead capability". As this requires decoupling the decode and execute stages, an instruction window or instruction pool gathers the instructions that are decoded and ready to issue.

▮ *Data Hazards*

Out-of-order completion creates output dependencies because instructions write their results in different order than they appear. There exists different techniques to alleviate those dependencies.

- Scoreboarding. This technique allows instructions to execute out-of-order when there are sufficient resources and no data dependencies. Every instruction goes through the scoreboard where a record of data dependencies is constructed. The scoreboard determines when the instruction can read its operands and begin execution. All hazard detection and resolution is centralized to the scoreboard. The MC88110 [AD92] implements scoreboard mechanisms to handle output dependencies.

- Tomasulo's Approach. Named after the scientist who invented it, this method combines key elements of the scoreboarding scheme with the introduction of register renaming. Register renaming eliminates name dependencies thereby handling WAR and WAW hazards. The register names are renamed with those from a larger virtual set of registers. This also helps in handling RAW hazards by internal forwarding which is not provided by scoreboarding. Tomasulo's scheme provides this functionality through reservation stations which buffer operands of an instruction waiting for issue. These reservation stations also pass results directly to waiting functional units instead of to registers as in scoreboarding. Tomasulo's algorithm implements register renaming by associating a tag to each register instance. This tag is then used in place of the register identifier. Appropriate tag management ensures in-order writing in each register. The IBM RS/6000 [Gro90] uses a variant of Tomasulo's algorithm to rename load destination floating point registers.

Other more complex but more powerful implementations use some type of buffering device which provides extra storage for instructions' results. An example of such device is the reorder buffer, a FIFO queue where issued instructions place their results. While instructions complete out-of-order, the buffer reorders the results so that they can be written in strict program order to the register file.

Out-of-order issue creates **anti-dependencies** because register values are not accessed in program order. Anti-dependencies are enforced by identifying the semantic instances of source registers at decode stage and associate them with the instructions in the instruction pool. Operands are thus copied with the instructions in the instruction pool in the form of values (if they are ready), register number (scoreboarding case) or tags(register renaming case). The mechanism must ensures that the operands of an instruction will not be overwritten and that the instruction will eventually issue when the proper operand values are ready and forwarded.

2.2.4 Committing State

The final phase of the lifetime of an instruction is the *commit* or the *retire* phase, where the effects of the instruction are allowed to modify the logical process state. The purpose of this phase is to implement the appearance of a sequential execution model even though the actual execution is very likely nonsequential, due to speculative execution and out-of-order completion of instructions. The actions necessary in this phase depend upon the technique used to recover a precise state. There are two main techniques used to recover a precise state, both of which require maintenance of two types of state: the state that is being updated as the operations execute and other state that is required for recovery.

In the first technique, the state of the machine at certain points is saved, or *checkpointed*, in either a *history buffer* or a checkpoint. Instructions update the state of the machine as they execute and when a precise state is needed, it is recovered from the history buffer. In this case, all that has to be done in the commit phase is to get rid of history state that is no longer required.

The second technique is to separate the state of the machine into two: the implemented physical state and a logical (architectural) state. The physical state is updated immediately as the operations complete. The architectural state of the machine is updated in sequential program order, as the speculative status of operations is cleared. With this technique, the speculative state is maintained in a reorder buffer; to commit an instruction, its result has to be moved from the reorder buffer into the architectural register file (and to memory in the case of a store), and space is freed up in the reorder buffer.

2.3 Case Studies

In this section we discuss two current superscalar microprocessors in the light of the microarchitecture described above. They are chosen to cover the spectrum of superscalar implementations as much as possible. They are DEC Alpha 21164 [EBea95] and UltraSPARC-I [web95].

2.3.1 Alpha 21164

The Alpha 21164 (Fig 3) is an example of a simple superscalar processor that forgoes the advantages of dynamic instruction scheduling in favor of a high clock rate. It has a quad-issue, superscalar instruction unit. Instructions are fetched from an 8 Kbytes instruction cache four at a time. These instructions are placed in one of two instruction buffers, each capable of holding four instructions. Instructions are issued from the buffers in program order, and a buffer must be completely emptied before the next buffer can be used. This restricts the instruction issue rate somewhat, but it greatly simplifies the necessary control logic.

- Pipeline Organization. The Alpha 21164 pipeline length is 7 stages for integer execution, 9 stages for floating-point execution, and as many as 12 stages for on-chip memory instruction execution. Additional stages are required for off-chip memory instruction execution.

- Branch Prediction. The branch prediction logic predicts conditional branch instructions using a branch history table with 2K entries addressed by low-order 11 bits of the PC. Each is a two-bit counter that increments when branches are taken and decrements when branches are not taken. The counter saturates at the top and bottom counts. A branch is predicted to be taken if the current

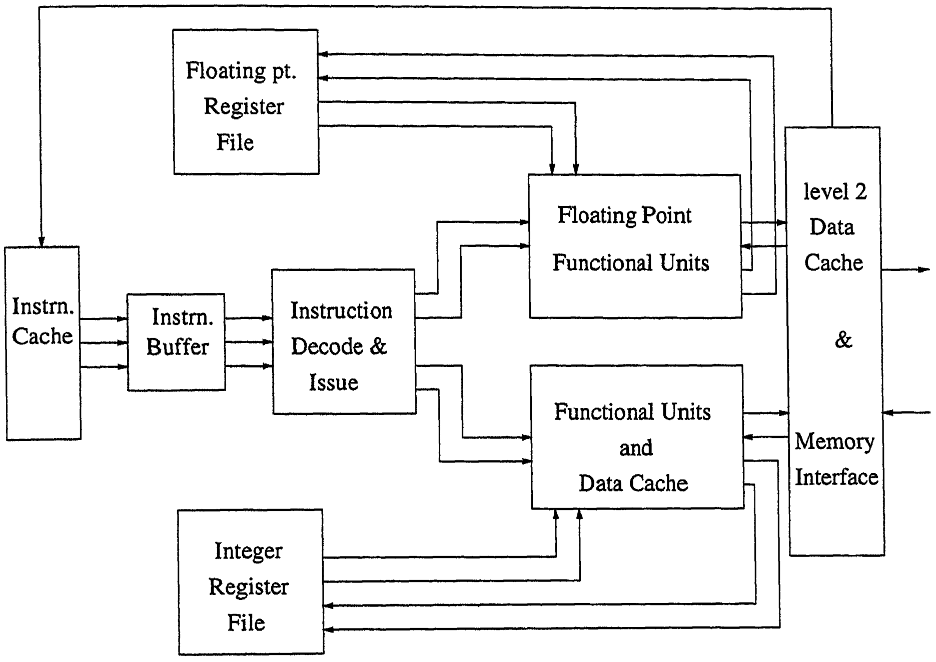
Floating pt. Register File
Floating Point Functional Units
level 2 Data Cache & Memory Interface
Instrn. Cache
Instrn. Buffer
Instruction Decode & Issue
Functional Units and Data Cache
Integer Register File

Figure 3: DEC Alpha 21164 Superscalar Organization

counter value is one of the two highest counts; otherwise, it is predicted to be not-taken. This method is more effective than the method used in the first Alpha microprocessor (i.e., Alpha 21064 [Sit93] [McL93]) which had only one bit of history per entry, partly because it reduces the misprediction rate for typical loop branches by half [EBea95].

- Instruction Issue. Following instruction fetch and decode, instructions are inspected and arranged according to type, i.e., the functional unit they will use. Then, provided operand data is ready (in registers or available for bypassing) instructions are issued to the units and begin execution. During this entire process, instructions are not allowed to pass one another.

- On-chip Caches. The 21164 has two levels of cache memory on the chip. There are a pair of small, fast primary caches of 8 Kbytes each—one for instruction and one for data. The secondary cache, shared by instructions and data, is 96Kbytes and is three-way set associative. The small direct mapped primary cache is to allow single-cycle cache accesses at a very high clock rate.

Many workloads benefit more from a reduced latency in the data cache than from a large data cache. The designers considered a single-level design for a large data cache. For circuit reasons, physically large caches are slower than small caches. To achieve a reduced latency, they chose a fast primary cache backed by a large second-level cache. As a result, the effective latency of reads is better in the Alpha 21164 CPU chip than it would have been in a single-level design [EBea95].

The two-level data cache has other benefits. The two-level design makes it reasonable to implement set associativity in the second-level. Set associativity enables power reduction by making data set access conditional on a hit in that set. The two-level design also allows the second-level cache to hold instructions, which makes a larger instruction cache unnecessary [EBea95].

The performance of the new implementation results from aggressive circuit design using 0.5 micron CMOS technology (Alpha 21064 uses 0.75 micron technology) and significant architectural improvements over the first Alpha implementation. The chip is designed to operate at 300 MHz, an operating frequency 10 percent faster than the previous implementation (the DECchip 21064) would have if it were scaled into the new 0.5 micron technology [EBea95]. The key improvements in machine organization are a doubling of the superscalar dimension to four-way superscalar instruction issue; reduction of many operational latencies, including the latency in the primary data cache; a memory subsystem that does not block other operations after a cache miss; and a large, on-chip, second-level write-back cache.

2.3.2 UltraSPARC-I

UltraSPARC-I is a highly integrated 64-bit, four-way superscalar processor targeted at running real life applications 2.5-5X faster than the previous SPARC processors [TGN95].

- Pipeline Organization. UltraSPARC-I uses a double-instruction-issue integer pipeline with nine stages: fetch, decode, grouping, execution, cache access, load miss, integer pipe wait, trap resolution, and writeback. These stages imply that the latency (time from start to end of execution) of most instructions is nine clock cycles. However, at any given time, as many as nine instructions can execute simultaneously, producing an overall rate of execution of one clock per instruction in many cases. In reality, some instructions may require more than one cycle to execute due to the nature of the instruction, a cache miss, or other resource contentions.

- Branch Prediction. Every cycle, up to four instructions can be prefetched from the instruction cache and sent to the instruction buffer. Each line in the I-cache contains eight instructions (32 bytes). Every pair of instructions has a 2-bit branch prediction field which maintains history of a possible branch in the pair. The four prediction states are the conventional strongly taken, likely taken, strongly not taken, and likely not taken. The advantage of the in-cache prediction scheme is that it avoids the alias problems encountered in branch history buffer. Implemented in this way, every single branch in the I-cache has its dedicated prediction bits, which translated into a high successful prediction rate of 88% for integer code, 94% for floating-point (SPEC92), and 90% for typical database applications [TGN95].

- Instruction Issue. UltraSPARC-I dispatches and executes instruction in the same order as they appear in the code. Every cycle, the grouping logic dynamically computes how many of the top four instructions sitting in the instruction buffer can be dispatched to the functional units. Instructions are allowed to complete out-of-order so that long latency operations can be bypassed by shorter ones. For instance, loads to the data cache, to the external second level cache, or to main memory are allowed to finish out-of-order with respect to other classes of instructions such as integer operations, floating point operations, etc. Thus, the UltraSPARC-I execution model, while maintaining the simplicity of an in-order execution model, takes advantage of a nonblocking memory system and scoreboarded loads in order to hide load latency operations.

- On-chip Caches. The UltraSPARC-I Data Cache is a 16 KB direct mapped, software selectable write-through non-allocating cache that is used on Load or Store accesses from the CPU to cacheable pages of main memory. It is a virtually-indexed and virtually-tagged cache. The D-cache is organized as 512 lines with two 16-byte sub-blocks of data per line. Each line has a cache tag associated with it. On a D-cache miss to a cacheable location, 16 bytes of data are written into the cache from main memory. For UltraSPARC, a significant advantage is gained through the use of a virtual cache [web95]. Physically tagged caches require the cache to be compared with the physical address from the MMU. In such environments cache miss detection may be a bottleneck in the MMU. While physical caches are appropriate for certain environments, virtual caches, such as those employed in UltraSPARC-I, increase performance through accelerated cycle cache miss detection.

  The Instruction Cache is a 16 KB, two-way set-associative cache used on instruction fetch accesses from the CPU to cacheable pages of main memory. Like the D-cache, it is virtually indexed and virtually tagged. It is organized as 512 lines of 32 bytes each, with each line having an associated cache tag. On an I-cache miss to a cacheable location, 32 bytes of data are written into the cache from main memory.

  The UltraSPARC-I External Cache Unit (ECU) efficiently handles instruction and data cache misses, handling one access per cycle to the external cache. These accesses are pipelined, consume only 3 cycles, and return 16 bytes of instructions or data per cycle. This effectively makes the E-cache an extension of the pipeline, allowing programs with large data sets to obtain data they need from the E-cache with load latencies that are based on E-cache latency. Floating-point application can make specific use of this feature to hide D-cache misses. UltraSPARC-I supports a variety of external cache sizes ranging from 512 KB to 4 MB. A MOESI (modified, own, exclusive, shared, invalid) protocol is used to maintain coherency across the system.

- Multimedia Support. Graphics speed has a big effect on a workstation user's perception of performance. Until recently, specialized graphics hardware was required for these operations. Typically, additional functionality could be provided to the base machine by adding one or more graphics cards. UltraSPARC-I designers saw an opportunity to provide a standard multimedia capability for future SPARC systems with only a 3% increase in die area [TGN95]. Multimedia instructions are executed by two specialized execution units in the floating point datapath. These RISC style instructions provide the core operations needed by multimedia algorithms.

The realization of UltraSPARC-I required 5.4 million transistors and the circuit was designed using 0.5 micron CMOS process technology. The chip is designed to operate at 167 MHz.

2.4 Java Chips

Java is a general-purpose, object-oriented computer programming language that offers special features that allow programs to take advantage of the power and flexibility of the Internet. Its cross-platform compatibility is motivating some software companies, such as Corel, to develop large-scale business applications entirely in Java [Way96].

Sun Microsystems, the company that launched Java, is betting on dedicated Java chips to deliver the performance needed for Java-based business and embedded applications. To this end, Sun is developing a core specification— known as picoJava—for Java chips. Chips based on Sun's picoJava core architecture should appear early in '97 and make their way into commercial products by the end of the year.

Sun's picoJava architecture will be the foundation for the first-generation Java chips, known as microJava, a low-cost family for resource-stingy embedded applications like PDAs, cellular phones etc. Sun is also developing a more expensive chip called ultraJava which will be for desktop systems. The architectural specification (i.e., picoJava) outlines a number of design innovations for optimally running Java code.

2.4.1 Stack Usage

What makes picoJava chips different from other processors? Foremost is how picoJava refines the stack. In the picoJava architecture, Java chips allocate variables locally on the stack, and method calls and bytecode operations also pass data through the stack.

Most C compilers convert C source code into a stack-based language, but the compilers then go through an additional step of converting this intermediate language into native RISC code. This allows the compiler to analyze the flow of data and keep the most essential elements in the CPU registers. A standard RISC processor simulates a stack machine by loading or storing data from the stack into registers, then using one of the registers to represent the stack pointer. This operation is simple, but the number of registers limits the opportunities for optimization [Way96].

The picoJava architecture uses a stack of sixty-four 32-bit registers with a pointer to the top register on the stack. If, for example, 20 registers are allocated for a particular stack frame (call it method A), then a call to another method, say B, would begin using register 21. The pointer or the top of the stack would move down from 20 to the last register used by method B.

2.4.2 Smart Cache

Sun architects devised a clever method of caching data if all the registers are full. For example, when method B is invoked, the picoJava register file allocates all remaining empty registers and carries over to register 1 if additional space beyond 64 is required. Something called the "dribbler" steps in from the background to restore the method-A data when method B finishes. The dribbler constantly reads and writes data from the 64 registers to a copy that's kept in memory. So when method B grabs the additional registers, the dribbler has already copied the data. If for some reason the dribbler hadn't yet made a copy, the Java chip would pause any processing tasks until the dribbler finished this operation. When method B stops running and gives up the registers, the dribbler restores the data to the stack, making method A current.

The dribbler takes advantage of the fact that the data traffic between the registers and its image in memory is highly predictable. System designers are able to easily tune a cache to anticipate the requests of the dribbler and make sure that necessary data is available in the local data cache when it needs to be.

The flexible register approach of picoJava contrasts with the simple register files of RISC processors. Java's dribbler dynamically tries to keep all the local variables available in fast registers. RISC chips, on the other hand, rely upon the compiler to orchestrate the movement of information in and out of the chip. Static register allocation works well with scientific code, which may have complicated loops that use each piece of data in multiple calculations.

The picoJava architecture also overcomes the inherent disadvantage of stack machines. Stack architecture typically use two instructions to move a variable to the top of the stack and then perform an add computation. PicoJava issues the move and the arithmetic operations together so they execute at the same time without disturbing the stack, writing over a register, or forcing the dribbler to do something. This reduces memory accesses and potentially cuts execution time.

2.4.3 PicoJava Pipeline

RISC pipelines driven by optimizing compilers have done quite well, and so Sun uses a very RISC-like pipeline for picoJava. The pipeline has only four stages: fetch, decode, execute and writeback. Sun is hoping that an innovative stack architecture and a stripped-down pipeline design will add up to fast performance for picoJava chips.

However, not everyone believes dedicated Java chips are necessary. After all, university researchers have built specialized chips for languages such as Lisp or Smalltalk only to discover that software implementations running on RISC chips offered superior performance. Also some chip vendors say their existing RISC and CISC architectures can handle Java quite well.

A description of picoJava finds its place here in the report because of it embracing the stack architecture, it being a pipelined RISC processor and most importantly, the architectural innovativeness of using a smart cache to offset memory latency.

Chapter 3

Instruction Set Architecture of PERL RISC

This chapter gives the instruction set of **PERL RISC**. We designed **PERL RISC** to be a memory-to-memory architecture because of its relevance today. The increase in bus bandwidth (both internal and external to the processor) available with current day microprocessors also supports our design.

3.1 Instruction Types

PERL RISC is a memory-to-memory architecture as we have no general purpose registers. The memory addresses of operands are themselves available as part of the instructions. Figure 4 specifies the format of instructions. Appendix C gives the instruction set of **PERL RISC**. All instructions in the instruction set have 3 operands except for unconditional jumps and traps whose format is also given in the appendix. The number of operands for all classess of instructions is specified in appendix C.

The instruction set is indeed one of the key features of computer architecture, defining and describing the capabilities of any computing system, including microprocessors [Tab94]. It constitutes a specific set of operations that a given system can perform. Following types of instructions are supported:

- Integer arithmetic and logical instructions.
- Shift and rotate instructions.
- Control transfer instructions.
- Floating point instructions.

We don't have data movement instructions (as they can be emulated using integer arithmetic) and Load/Store instructions as it is a memory-to-memory architecture. Further each instruction can be data typed making it possible to convert from one data type to another at the time of implicit memory load/store.

3.2 Addressing Modes

In this section, we look at the addressing modes—how architectures specify the address of an object they will access—supported by **PERL RISC** architecture. Any RISC machine supports only a minimum number of addressing modes—about 4. We have strictly struck to this RISC philosophy of fewer addressing modes keeping in mind decoding complexity which will have a telling effect on pipeline design. **PERL RISC** supports the following addressing modes:

1. Base Addressing. This mode is primarily useful for accessing the local variables on the stack. First four memory locations (addresses 1 ...4) are used to represent base addresses. In the instruction encoding, 2 bit are allocated for each of the three operands to indicate which location is used as the base address. The compiler for C [Kum97] on this machine, however uses only the first 2 locations, one for stack pointer (SP) and another for frame pointer (FP).

2. Direct Addressing. In this case, the instruction itself has the address of the operand. This is taken as the Effective Address of the operand.

3. Memory Indirect. This is basically used for accessing array elements.

4. Immediate. In this case the value of the operand is part of the instruction.

5. PC Relative. Jump addresses can be either absolute addresses or PC relative. The compiler may generate a PC relative jump by specifying the offset in the instruction. PC is added to this value to get the destination address.

Table 1 lists the addressing modes supported. If addressing mode indicator bits

Addressing Mode	Value (2 bits) [1]
Base/PC relative	0
Direct	1
Memory Indirect	2
Immediate	3

Table 1: Addressing Modes for Operands

are 00, it indicates base addressing mode for arithmetic and logical instructions, but for jumps, it indicates PC relative target address.

3.3 Data Types

PERL RISC is a 64-bit architecture as we support 64-bit data types. Table 2 gives the list of data types supported. Out of the 3 bits, the least significant two

Data Type	value(3 bits) [1]
unsigned byte1(8 bits)	0
unsigned byte2(16 bits)	1
unsigned byte4(32 bits)	2
unsigned byte8(64 bits)	3
signed byte1(8 bits)/float4(Single Precision FP)	4
signed byte2(16 bits)/float8(Double Precision FP)	5
signed byte4(32 bits)	6
signed byte8(64 bits)	7

Table 2: Data Types of Operands

[1]This is the value used while encoding instructions

bits represent the data type. The most significant bit of the 3 bits indicates whether the arithmetic is signed or unsigned. If it is 1, it is signed arithmetic. If data type value is 4 or 5, the opcode indicates whether the arithmetic is integer or floating point.

3.4 Instruction Format

All the instructions are encoded in the fixed width of 128 bits. Figure 4 gives the format of each instruction. The first 32 bits of the instruction are used to specify the addressing modes and data type of the operands apart from the opcode. It should be clear from the encoding

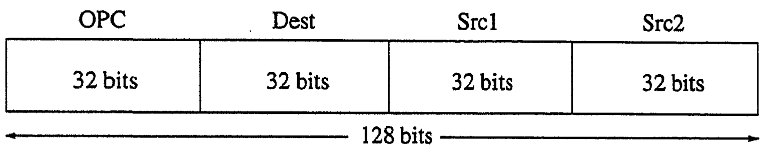

Figure 4: Instruction Format

mechanism that there is no restriction on the type of addressing mode for operands i.e, each of the operands can be specified using any of the 4 addressing modes. If the destination addressing mode is immediate, the result is not stored, even though the computation is carried out.

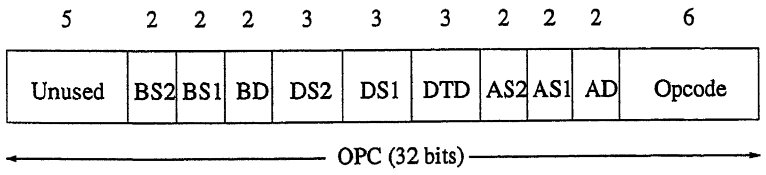

Figure 5: Instruction Encoding

The next three 32-bit words are used to indicate the address of each of the three operands. Figure 5 gives the instruction encoding mechanism. The format specifies the addressing mode and data type indicator bits.

The meaning of each field in Figure 5 is explained below:

- Opcode. These 6 bits refer to the opcode of the instruction. Refer Appendix B for a list of instruction mnemonics and their opcodes. We have allocated only 6 bits for opcode because we have only those instructions which are absolutely essential. For example, most machines provide a NEG (negate) instruction which gives the 1's compliment. This instruction is not necessary because the same effect can be obtained using an XOR instruction.

- AD, AS1 and AS2. Each of these fields is 2 bits. These indicate the addressing mode of the three operands, namely, destination, source 1 and source 2 respectively. The assembler encodes the addressing mode bits for operands based on the values given in Table 1.

- DTD, DS1, DS2. Each of these fields is 3 bits. These indicate the data type of the three operands, namely, destination, source 1 and source 2 respectively. The assembler encodes the date type for operands based on the values given in Table 2. As the C compiler [Kum97] generates code where all the operands of an instruction are of same type, the simulator uses DTD bits as the data type of all the operands, but this flexibility is provided for compilers which can use it effectively. One advantage of providing this flexibility, of course, is that it obviates the need for data type conversion instructions which are provided by most machines to perform implicit type conversion.

- BD, BS1 and BS2. If the addressing mode of an operand is based, the corresponding base address is given by these fields. These fields are looked at during decoding only if the addressing mode indicator bits specify using base addressing mode.

- Unused. These bits are not used in the current implementation.

Chapter 4

Pipeline Design for PERL RISC

As explained in chapter 2, pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques, it is transparent to the programmer. This chapter gives the design of **PERL RISC** pipeline.

4.1 Pipelining—Issues

The *throughput* of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. Time required for moving an instruction one step down the pipeline is a *machine cycle*. Because all stages proceed at the same time, length of the machine cycle is determined by the time required for the slowest pipe stage. A designer's goal is to balance the length of the pipeline stage. If stages are perfectly balanced, then the time per instruction on the pipelined machine—assuming ideal conditions (i.e., no stalls)— is equal to

$$\frac{\text{Time per instruction on non-pipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages (i.e., if there are n stages then the speedup is n). Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible, though it can be close (within 10%) [PH94].

Pipelining yields a reduction in the average execution time per instruction. This reduction can be obtained by decreasing the clock cycle time of the pipelined machine or by decreasing the number of clock cycles per instruction, or by both.

4.2 The Different Pipeline Stages

PERL RISC pipeline has the following stages:

Fetch: This stage puts the instruction at the tail of the instruction queue.

Decode: This stage reads instruction from the head of the instruction queue, decodes them and places them in the instruction window of the integer or the floating point operational unit.

Issue: checks for availability of operands. When they are available, the instruction is ready for issue.

Write Back: the instruction's completed result is written in the reorder buffer.

Commit: the instruction's result is written to the memory location. The commit point is the time when the instruction modifies the processor state.

Figure 6 shows the different pipeline stages. Even though we have highlighted only 5 stages above, the figure shows additional stages where the cache is accessed. We have a seven stage pipeline if the accesses to the cache are included. In fact, before the IF stage, there is an access to the Instruction cache which is not shown in the figure.

4.2.1 Fetch Stage

This stage fetches instructions from the I-cache and places them in the instruction queue. The presence of branches creates two major problems that disrupt the fetch mechanism:

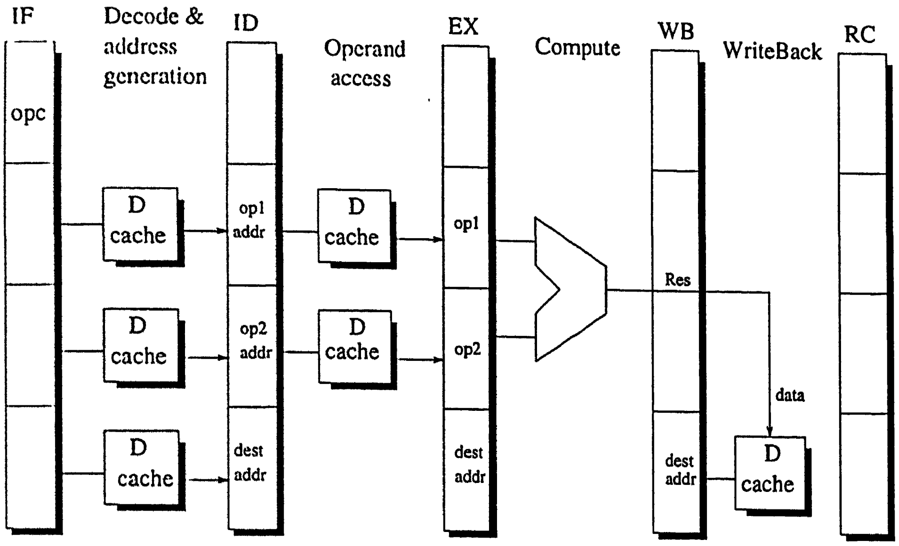

Figure 6: Processor Pipeline

1. Conditional branches hinder fetch as the PC value for the next instruction to be fetched depends upon the branch outcome which is not known during fetch. execution;

2. Instructions are no longer aligned along cache boundaries due to branch. This is because if the branch instruction is present in the middle of a cache block those instructions to the right side may not be valid. Similarly targets of branches may not be aligned to the left hand side of a block.

The first problem is resolved by branch prediction: instead of stalling when it recognizes branches, the fetch process performs branch prediction based on the past behavior of the branch recorded in the branch target buffer. The branch prediction mechanism used is described in chapter 5. The second problem is not modeled in the simulator; instructions are fetched based on their own PCs and thus we do not consider them as parts of fixed sized cache blocks which could contain non valid instructions. As a consequence instruction alignment is ensured.

The branch prediction mechanism used helps in reducing the negative effect of conditional branch on the fetch process. Direct jumps are handled easily as their targets can be calculated at fetch stage.

However the presence of indirect jumps create more problems. The fetch process has to stall when it encounters an indirect jump because the value of the jump location may not have been computed. As the fetch stage is stalled, the stall propagates through the pipeline thereby increasing the branch penalty.

4.2.2 Decode Stage

This stage takes instructions from the instruction queue, decodes and dispatches them to their appropriate operational unit, integer or floating point. The maximum number of instructions decoded per clock cycle is a simulation parameter. Each decoded instruction is assigned an entry in the reorder buffer of its operational unit where it is placed along with its destination location identifier.

The entry is assigned in program order (which is the decode order) at the tail of the reorder buffer which acts as a FIFO queue. The point is that all instructions go to the reorder buffer, even those who don't need to take part in the renaming process.

As a consequence dummy entries which do not produce any results for the memory location can be created in the reorder buffer; this simplifies the processing of branches as we will see in chapter 5.

Once a reorder buffer entry is created for an instruction, it enters the enters the *central instruction window* of its operational unit from which it gets issued. To check for data dependencies, values of the instruction's operands should also be placed in the window entry. To do so, the operand address is searched for in the reorder buffer. If it is available there, but is not valid (i.e., it has not been computed), the corresponding reorder buffer entry number is taken in place of the operand value. If the operand is available in the reorder buffer and is valid then the computed value can be taken. Provision is made to get the most recent value entry in the reorder buffer. Finally, if this operand is not found in any entry of the reorder buffer, then its value is read from memory.

Actually, the decode stage is implemented as two different stages in the pipeline because of the indirect and based addressing modes which **PERL RISC** supports. The two stages are:

1. Address Computation. This stage actually computes the effective address of operands. Operands which are accessed using immediate mode and direct mode are ignored because their effective address is already available. For indirect and based addressing, we calculate the effective address. As address forwarding is also done, we search the reorder buffers in this stage also.

2. Operand Access. At this stage, the address of operands are available. This is used to read the operand value. We search the reorder buffers and do the appropriate action specified in the above paragraph.

4.2.3 Execute Stage

The issue logic selects the instructions which have operands and functional units available by examining the instruction window. When two instructions conflict for the same functional unit, the selection is made based on the age of the instruction in the window: the oldest one has the highest priority. The implementation of the central window as a compressible stack helps in prioritizing instructions. As instructions can enter only at the top, the older instruction lies at the bottom of the stack.

Latency of various functional units is simulated: when a long latency operation is issued, its result is calculated at once and placed in the reorder buffer entry of the instruction with a ready field indicating at which clock cycle the result is available. If a branch prediction happens to be incorrect, the instructions following the branch in both reorders buffers are flushed. In this case, the branch target buffer is updated with new prediction information.

4.2.4 Write Back Stage

This stage identifies the reorder buffer entries whose results have been computed and validates them. These results are also forwarded to other instructions which need them. The Write Back logic also frees the corresponding functional unit.

4.2.5 Result Commit Stage

The results that have been validated during the Write Back stage are sent to the corresponding memory locations during this stage. The writes are processed in order from the head to the tail of the reorder buffer until an instruction is found with an incomplete result. The committed instructions are removed from the reorder buffer. Invalidated instructions that follow a wrong branch prediction are simply discarded.

4.3 Pipeline Hazards

Even though, pipelining offers obvious advantages, it comes with its own problems. In chapter 2 we mentioned some of the hazards that create problems for out-of-order execution. This section explores them in more detail. There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards lower the performance from the ideal one gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

2. *Data hazards* arise when an instruction depends on the result of a previous instruction in a way that is exposed by overlapping of instructions in the pipeline.

3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

4.3.1 Structural Hazard

The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of all instructions which use this functional unit cannot be initiated sequentially in the pipeline. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a machine may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle.

4.3.2 Data Hazards

Data hazards occur when the pipeline changes the order of accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined machine. A simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*) can be used to minimize stalls due to data hazards. In this technique, ALU result from the Execute stage is always fed back to the ALU input latches. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i occurring before j. The possible data hazards are:

- RAW (read after write)—j tries to read a source before i writes it, so j incorrectly gets the old value. This is the most common type of hazard and the kind we used forwarding to overcome in **SuperSIM**.

- WAR (write after read)— j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline. For example, autoincrement addressing can create a WAR hazard. This cannot happen in **PERL RISC** pipeline as all reads are early (in ID stage) and all writes are late (in WB stage) and also because instructions commit in order.

- WAW (write after write)—j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard is present in our pipeline as we allow instructions to proceed even when a previous instruction is stalled. The proper ordering is ensured by the use of reorder buffers, which is explained in detail in chapter 5.

- RAR (read after read)— this case is not really a hazard.

4.3.3 Control Hazards

Control hazards can cause a greater performance loss for our pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. The method we adopted to reduce the branch penalty is to use a 2-bit branch prediction scheme which is also explained in detail in chapter 5.

Chapter 5

Simulator Implementation

Simulation techniques are widely used for the design of a new processor. A trace driven simulation is fast but does not execute the target program which cannot ensure the correctness of the simulation. An instruction set simulation is a technique for simulating the exact behavior of the processor by simulating each instruction and getting the performance metrics.

5.1 SuperSIM–The Superscalar simulator for PERL RISC

SuperSIM, the superscalar simulator for **PERL RISC** is an instruction set simulator. We took this "execution" oriented approach for different reasons:

- Correctness of Results

  We wanted the results of simulated assembly instructions to be effectively computed and the contents of the different hardware elements to be recorded on a cycle basis. Correctness of simulated program output was very important for us to assess the proper coordination of all the different simulated hardware components.

- Multi-Platform Usage

We wanted a simulator executing assembly programs generated by a cross compiler, so that it be portable to different machines.

- Availability

  The availability of simulators for existing architectures. DLX [PH94] is an hypothetical architecture and lot of work has been done on it. *DLXsim* [LB92] and *SuperDLX* [Mou93] are two such simulators. We made extensive use of these program segments wherever necessary and also made the necessary modifications so that the new simulator which we had written simulates **PERL RISC** architecture.

SuperSIM implements the most sophisticated superscalar instruction processing policy: multiple out-of-order issue, multiple out-of-order completion. To achieve this policy efficient hardware mechanisms were selected for simulation, such as, a central window buffering instructions for issue, and a reorder buffer, supporting register renaming. Other features, like branch prediction were built around them to go even further in performance.

5.2 Underlying Data Structures

The simulator needs to model two types of components in the processor:

- Memory and the functional units; these are the fundamental components of the processor.

- Other machine components like the branch target buffer (BTB), the instruction queue, the instruction windows and reorder buffers. We collectively call them the *superscalar hardware* elements. This grouping is not quite appropriate because using a branch target buffer along with an instruction queue is a strategy which is also used in non-superscalar processors to improve the efficiency of fetch mechanism

The parameters for configuring the machine are specified in the machine configuration file which is read by the simulator before the simulation starts. An example machine configuration file and its contents are given in appendix B.

5.2.1 Instruction Tables

The opcode values in the instruction set are used as indices to the *opcode description table* of the simulator. Using this table we can obtain characteristics of an instruction at each pipe stage and identify its behavior. This table reflects all the choices that were made concerning the processing of each instruction in the superscalar processor:

- choices on the functional unit corresponding to an instruction. By default, the instructions are sent to the operational unit that correspond to their result type.

- choices on the handling of special instructions such as traps.

The reason behind using this table is to gather instructions into groups that have the same characteristics.

5.2.2 Basic Processor Elements

▮ *Memory*

Memory is implemented as an array. Accessing the memory locations is through the use of indices (which are the memory location addresses, addresses of data and PCs of instructions). The memory size is a parameter.

The memory locations, (wordPtr), can contain either a value or an instruction, and consist of the following fields:

- value: content of the memory location (for data accesses).

- opcode: opcode (for instructions), number which permits to access the opcode description table; it is initialized to NOP.

- src1, src2, dest: The 32 bit addresses of operands.

Even though a memory location should have only "value", it is modeled as an array of structures consisting of the above fields for simulation purposes. As an example, the opcode, which is actually encoded in "value" field of the first word of an instruction, is copied to the opcode field of the structure when the instruction is first encountered as it is used repeatedly during every stage. It is used in the fetch stage for branch prediction and in the decode stage to find the functional unit to which it belongs. Without a separate field, bit manipulation should be done every time it is needed.

▮ *Functional Units*

Two tables represent the functional units in the Simulator:

- intUnits[MAXUNITS]: integer functional units
- fpUnits[MAXUNITS]: floating-point functional units

The entries in this table correspond to a functional unit type which has the following configurable parameters:

- latency: This corresponds to the number of cycles that elapse between issue and write back of an instruction. This is used to fill the ready field of an instruction in the reorder buffer, when this instruction is issued; it is a parameter which can be specified in the machine configuration file.
- num units: number of available instances of that type of functional unit; it is also parameterized.
- num used: number of functional units of that type which are currently used; this permits the resolution of structural hazards.

Each functional unit type information is indexed by its identifier. Functional unit identifiers are associated to instruction types in the opcode description table, so that the instruction issue logic knows where to send instructions at issue time.

5.2.3 Superscalar Elements

This section details the mechanisms used to support the superscalar aspects of superSIM.

▮ *Branch Target Buffer*

SuperSIM uses the 2-bit branch prediction scheme which is widely used by current day superscalar microprocessors. This mechanism helps the fetch stage to predict the outcome of a conditional branch based on its past behaviour and thus fetch from the predicted path.

SuperSIM uses a table to model the Branch Target Buffer. The index to the table is the lower order bits of branch addresses (i.e., address modulo size of the table). The fetch stage uses the contents of this table to predict a branch. The size of BTB can be specified in the machine configuration file. Each entry in the BTB contains the following fields:

- address: the address of the branch instruction for which the entry has corresponds to. This field is necessary to resolve conflicts between branches that have the same lower order bits.
- predictState: a two bit number giving the next predicted outcome for the branch. This number can take four values:
  - STRONGLY TAKEN(11), TAKEN (10)— which make the branch be predicted taken.
  - NOT TAKEN(01), STRONGLY NOT TAKEN (00)— which make the branch be predicted not taken.

When a branch instruction is encountered for the first time, the prediction is set to be STRONGLY_TAKEN. It is then updated every time the effective outcome of the branch is determined after executing the branch; taking the branch increments the prediction number, not taking it causes it to be decremented. We do not wrap around when the field reaches its maximum or minimum. Figure 7 shows the finite sate machine for the two bit prediction scheme. As we can see, a two bit scheme minimizes oscillation that could occur in a one bit scheme.

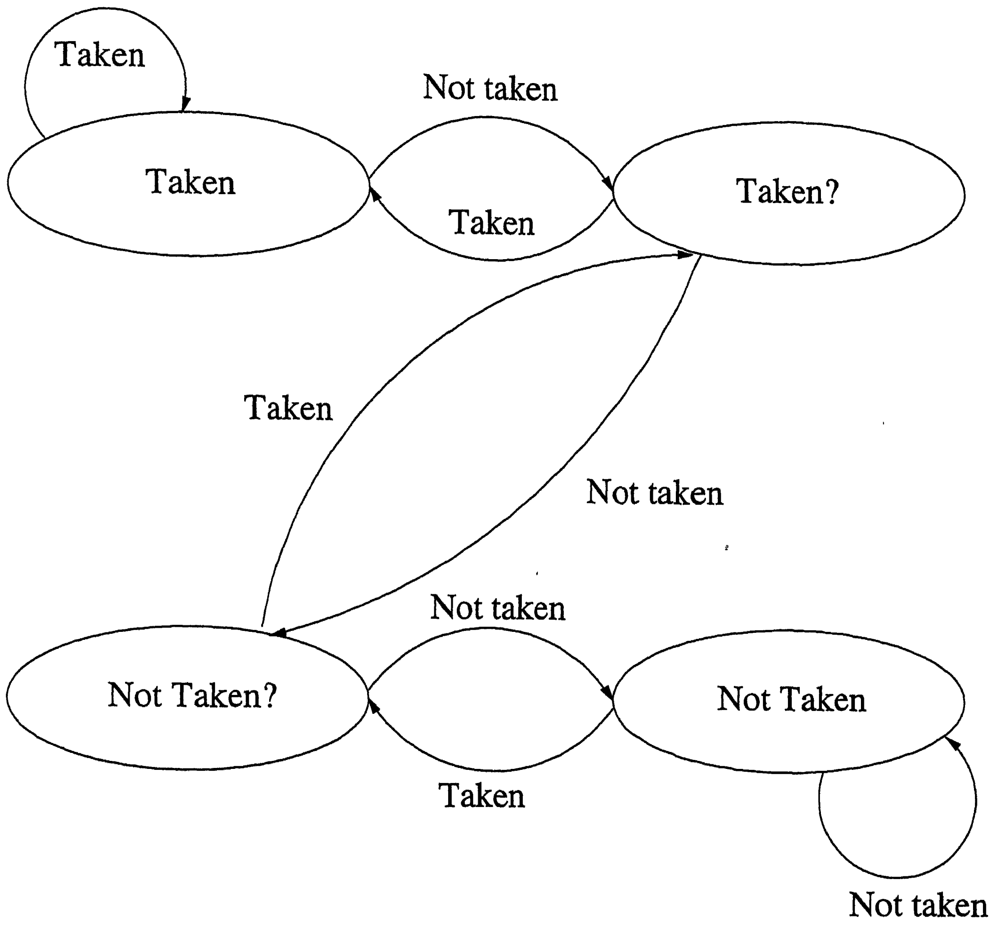

Figure 7: State Diagram for a 2-bit Branch Prediction Scheme

▌ *Instruction Fetch Queue*

The Instruction Queue is a FIFO queue where the fetch stage adds a fixed number of elements at the top, and the decode stage takes a fixed number of elements from the bottom. **SuperSIM** uses a doubly linked list to model the instruction queue because one stage adds entries to it and another stage deletes it. A doubly linked list is used because, for deletion, we need to know the address of the previous entry. Each item contains the following information:

- wordPtr: memory word fetched.
- address: instruction address (pc).

- prediction: prediction information for branches: TAKEN, NOT TAKEN. This is the prediction made by the fetch process when it encounters a branch. It is used only for branch instructions.

The maximum number of elements in the instruction queue (numEntries) is a user set parameter from the machine configuration file.

▮ *Instruction Windows*

There are two instances of these; *iWindow* for integer instructions and *fpWindow* for floating point instructions. Their sizes can be configured by specifying them in the machine configuration file. In the processor, instruction windows are compressible stacks that keep the instructions in order. Decoded instructions are placed at the top of the stack, but they can be issued from any position in the stack. Keeping instructions in order makes it simpler to prioritize among ready instructions, as older instructions appear at the bottom of the stack. But this requires hardware mechanisms to allocate and compress the window. The simulator simply models this mechanisms with linked lists.

The entries in the instruction window each correspond to an instruction that has been decoded, and contains the following fields:

- opCode: the decoded opcode of the instruction.
- class: the instruction class corresponding to the type of the functional unit.
- unit: the functional unit where the operation must take place; it is determined by the decode function (using the opcode description table).
- reorderEntry: the reorder buffer entry the instruction is assigned to. This link between the operation in the window and its corresponding entry in the reorder buffer must be kept for the issue logic to process the instruction properly.
- prediction: the prediction information (TAKEN, NOT_TAKEN) for branches.
- firstOperand: the information on the first operand:

  - value—which contains the operand value or the reorder buffer entry from where the value will be forwarded when computed.
  - type—which contains the type of the operand (integer, floating point double, floating point single).
  - valid—which indicates if the operand is ready to be used.

- secondOperand:the information on the second operand if it exists (same structure as for the firstoperand).

Reorder Buffers

Two instances of reorder buffers, iReorderBuffer and fpReorderBuffer, are modelled by the simulator. Their respective sizes are parameters which can be configured. Each of them is a circular queue, with a head position (bottom of the queue) and tail (top of the queue). New decoded instructions are assigned an entry at the tail of the queues, and they commit in order at the head. With this mechanism, instruction entries do not have to move towards the bottom or top of the queue: only the head and tail move, and thus the buffers' entry numbers can be used to rename the result memory locations.

Each entry in the reorder buffer consists of the following:

- dest: the result memory location. It is used when an operand is searched in the reorder buffer and it indicates which location to write to when the instruction commits at the head of the buffer.
- opCode: the opcode of the instruction.
- class: the instruction class which the instruction belongs to; the result commit and write back logics use this field to know what to do with the instruction.
- result[2]: the computed result of the instruction (two numbers for double floating points). Some instructions, like conditional jumps, produce no result.
- valid: the validity bit of the result, which indicates if the operation has been completed, and thus if the result is available.

- ready: it indicates at what clock cycle the computation of the result is going to complete.

- insCount: the dynamic instruction count calculated when the instruction was fetched.

- flush: the flushbit. It is set to 1 when the instruction follows a wrong branch prediction.

A list is maintained, which links reorder entries of currently executing operation. This list of pending operations (listOfExec) makes the work of the write back stage easier and faster.

Chapter 6

Simulation Results

This chapter compares the performance of **PERL RISC** with other popular RISC architectures of today. We have compared the results generated by **SuperSIM** with that of DLX [PH94] and that of DEC Alpha 21064. DLX [PH94] is a hypothetical RISC architecture which can be taken as a representative of all current day RISC machines. The results for DLX are obtained using **SuperDLX** [Mou93]. The performance metrics for DEC Alpha 21064 are obtained using the ATOM [Dig93] performance analysis tool which is part of the DEC OSF software kit. Atom is built using OM, a link-time code modification system. OM takes as input a collection of object files and libraries that make up a complete program, builds a symbolic intermediate representation, applies instrumentation and optimizations to the intermediate representation, and finally outputs an executable. The instrumentation routines can be customized for one's needs. But we used the routines which are provided with the ATOM distribution. There are tools to get cache performance, pipe stalls, instruction profiling etc. As we used Atom on our DEC 21064, which is 2-issue superscalar, we configured **SuperSIM** and **SuperDLX** to simulate instructions in 2-issue mode. The results for 4-issue are available only for **PERL RISC** and **DLX**.

One important thing that should be noted is that the performance metrics are provided for same programs when compiler optimizations were used and when they are not. The reason being that the C compilers for **SuperDLX** and DEC Alpha

21064 did optimizations whereas the C compiler [Kum97] for **PERL RISC** did not do any optimizations. The performance of RISC machines is greatly improved by compiler optimizations as indicated in the tables. Some of the optimizations done by these compilers are making address computation loop independent, loop unrolling etc. For medium sized benchmark programs, hand optimization was done for **PERL RISC**. The metrics that are presented in the tables are:

- IC: The dynamic instruction count.
- CC: Total number of clock cycles required to execute the program.
- PS: Number of cycles the pipe is stalled.

6.1 Benchmark Programs

The programs which were simulated are:

1. Permutation. This is a heavily recursive program which, given an array of n integers, prints all n! permutations. This program is chosen mainly to identify how **PERL RISC** performs vis-a-vis other RISC machines because we have all dependencies mentioned in appendix E which cause fetch stalls. Because the program is heavily recursive, fetch stalls due to case 1 and 2 occur more often. Table 3 provides the performance metrics for the 3 architectures. The results are for n equal to 5.

Architecture	*with compiler opt.*			*without compiler opt.*		
	IC	CC	PS	IC	CC	PS
PERL RISC (2 issue)	8947	11506	6593	12761	15773	9006
PERL RISC (4 issue)	8947	10078	7202	12761	14012	10030
DLX (2 issue)	15423	8709	894	18115	10226	950
DLX (4 issue)	16310	10514	5757	20083	11816	5742
Alpha 21064	9698	9572	2069	9683	12274	4681

Table 3: Comparison for Permutation Benchmark

Even though DLX and 21064 represent the same class of RISC architectures, there is a disparity in the number of instructions executed. We found out the reason to be the code generated by the compilers. The C compiler for DLX generates a "nop" instruction after every load, store and branch instruction. So, the delay slot is not used in most of the cases. This accounts for the extra instructions executed by DLX. Another interesting point to note is that the stall cycles increases for both **PERL RISC** and **DLX** if we configure the simulators to emulate a 4-issue machine. This is understandable because as more instructions are issued in one cycle, the dependencies between instructions within the window increases (as a larger window is looked at), hence there are more stalls. But the percentage increase is more for DLX, which is not able to issue more instructions in this example. The reason is that the "jal" (jump and link) instruction, which is used for procedure calls, results in a stall of 4 cycles whenever it is executed. As the program is heavily recursive, this instruction is repeatedly executed for the recursive procedure call. In fact, a 4-issue DLX performs worse than a 2-issue machine. The reason for this was found to be the size of the load and store buffers (i.e., the number of entries that can be present here). The performance of 4-issue DLX improved when the size of these buffers were doubled. For obtaining the above statistics, we configured them to hold 5 entries. When this was increased to 10, the number of clock cycles taken by the simulator to simulate the optimized code came down from 10514 to 8536 indicating that Load/Store overhead is the main bottleneck there. The loads and stores were at the entry and exit of procedure calls when the registers are saved and restored. But in case of **PERL RISC**, even though the stall cycles increases, The 4-issue version performs better than the 2-issue version for the same machine configuration, as it takes fewer clock cycles. This shows that **PERL RISC** gives more scope for multi-issue.

2. Matrix Multiplication. This is another program which we simulated using **SuperSIM**. Table 4 gives a summary of the metrics obtained. The size of all the matrices were 32x32. Again, in this example, we see a great disparity in the number of instructions executed for DLX and 21064. The reason here is

Architecture	*with compiler opt.*			*without compiler opt.*		
	IC	CC	PS	IC	CC	PS
PERL RISC (2-issue)	357421	347804	168084	1083234	588299	46138
PERL RISC (4-issue)	357660	278007	171190	1087656	515402	241694
DLX (2-issue)	688341	345275	1104	1487115	744664	1106
DLX (4-issue)	704448	344197	1120	1503391	743068	1118
Alpha 21064	459340	977566	656717	1082772	1535495	721258

Table 4: Comparison for Matrix Multiplication Benchmark

because of the "s8addq" instruction in Alpha which performs scaled addition.

```
s8addq s_reg1,s_reg2/imm,d_reg
```

The above instruction computes the sum of two signed 64-bit values. This instruction scales (multiplies) the contents of s_reg1 by 8 and then adds the contents of s_reg2 or imm. The result is stored in d_reg. Such a powerful instruction is provided to access the elements of an array. The address of element a_i, for example can be calculated within a loop by this single instruction, whereas in **DLX** and **PERL RISC**, we need one add and one multiply instruction. As the matrix multiplication program accesses the elements of two-dimensional arrays, this instruction in Alpha is very useful. This example shows that **PERL RISC** performs better than the other two machines and that it scales well with increase in superscalar issue.

3. Time table scheduler. Given a list of courses and preference for timing for allotting slots to the course and given a set of class rooms, this program uses a heuristic approach to get the best optimized output. This program was also used as one of the benchmarks. As the assembly code generated for this program is big (more than 3000 lines of code) manual optimization could not be done for **PERL RISC**. So table 5 presents the results only for the unoptimized code for all three machines.

   From table 5 it should be clear than except for instruction count, we obtain expected results. The reason for instruction count disparity is again the

Architecture	*without compiler opt.*		
	IC	CC	PS
PERL RISC (2-issue)	4065151	3687273	1566469
PERL RISC (4-issue)	4271332	3168330	1908607
DLX (2-issue)	7172897	3772762	176408
DLX (4-issue)	7545677	3672534	1293016
Alpha 21064	3060510	2885116	469223

Table 5: Comparison for Time-table Scheduler Benchmark

"s4addq" and "s8addq" instructions.

From the above statistics, it should be clear that **PERL RISC** performs better or at least as good as existing RISC machines. As the C compiler [Kum97] for **PERL RISC**, is a port of gcc it does not generate code for this memory-to-memory architecture based machine. The reason being that the gcc compiler is optimized for machines with many registers. It was initially difficult to port it to **PERL RISC** which had no registers. For more information on how it was done refer [Kum97]. For the results obtained for **DLX** and **Alpha 21064**, it should be clear that compiler optimizations give a great improvement in performance. It will be interesting to write a custom compiler for **PERL RISC** and then compare the results of simulating that code with those of existing machines.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

From the results obtained, it is clear that **PERL RISC** performs better or at least as good as existing processors. To get the best performance of this we should of course compare results generated by a compiler which is custom built for **PERL RISC**. This should provide better performance as many features provided by **PERL RISC**, like different data types for operands of same instruction, additional locations for use in based addressing mode etc., are not fully utilized by RLcc [Kum97].

7.2 Future Directions

SuperSIM was developed within the limited scope of a Master's thesis, so there is certainly a lot of room for improvement and modification of the simulator. The goal of this section is to give some ideas for future enhancements of the simulator. This list of proposals is not intended to be exhaustive.

7.2.1 Improving or adding superscalar features

The implemented branch prediction mechanism permits to start execution of one conditional path of a branch. It will be interesting to experiment branch speculation on both paths of the branch and compare the two branch processing techniques.

This type of branch speculation is certainly very complex to implement; buffering of instructions has to be provided to support execution of two conditional streams of instructions at a time, which implies some kind of replication of the reorder buffer and instruction window. Duplication of hardware elements has to stay within realistic limits.

7.2.2 Improving the simulator interface and usability

The on-line interface allows the user to interact with the simulator through the use of some display and execution commands. There are others interesting commands that could be added to give to the simulator a full debugging capability:

- commands to inspect memory locations. The commands would return values in format specified by the user (decimal, hexadecimal binary, floating point double, floating point simple, character, string ...). Getting a memory location's value in a context of a superscalar machine could mean getting the values of the current instances of this location in the processor.

- commands to modify memory locations. The user should be able to store numbers, like integers or floating point. Again, this may not work the same way as in a scalar simulator: storing a value in a location can affect all the current semantic instances of that location present in the machine (this is because we support operand renaming).

- commands to inspect machine characteristics. The user could display parameters from the machine configuration file, and also changing values, like the current size of a window/buffer, the maximum size of a window/buffer so far ...

- commands to modify machine configuration parameters.

- providing an X-window interface. This could make the simulator more interactive. Also, the simulator can be used as part of the Computer Architecture course to study the working of a superscalar machine.

- adding new traps. C library functions are compiled by RLcc [Kum97] into traps. Also, as there are not enough traps(*e.g., scanf, fscanf*), a lot of benchmarks could not be simulated.

- adapting RLcc [Kum97] to the simulator. RLcc generates code for a non-superscalar machine. It would be interesting to study what compiler optimizations the superscalar machine could exploit and implement them in RLcc.

Appendix A

User Manual

NAME

SuperSIM - Simulator for assembly programs of our Register-less RISC architecture.

SYNOPSIS

`supersim`

OPTIONS

[-f filename] [-bp] [-ms#]

`-f filename` machine description file specifying user parameters.

`-bp` Turn on the branch prediction scheme.

`-ms#` Specify the size of the memory.

DESCRIPTION

SuperSIM is an interactive program that loads assembly programs and simulates the operation of a register-less RISC computer on those programs. Once started, **SuperSIM** loops forever reading commands from standard input and printing results on standard output.

NUMBERS

Whenever **SuperSIM** reads a number, it will accept the number in either decimal notation, hexadecimal notation if the first two characters of the number are **0x** (e.g. 0x3acf), or octal notation if the first character is **0** (e.g. 0342).

ADDRESS EXPRESSIONS

Many of **SuperSIM**'s commands take as input an expression identifying a register or memory location. Symbolic expressions may be used to specify memory addresses. The simplest form of such an expression is a number, which is interpreted as a memory address. More generally, address expressions may consist of numbers, symbols (which must be defined in the assembly files currently loaded), the operators $*$, /, %, $+$, $-$, <<, >>, &, |, and $\uparrow$ (which have the same meanings and precedences as in C), and parentheses for grouping.

COMMANDS

SuperSIM provides the following application-specific commands:

asm *file file file* ...

Read each of the given *files.* Treat them as assembly language files for our machine, and load memory as indicated in the files. Code (text) is normally loaded starting at address 0x100, but the **codeStart** variable may be used to set a different starting address. Data is normally loaded starting at address 0x8000, but a different starting address may be specified in the **dataStart** variable. The return value is either an empty string or an error message describing problems in reading the files. A list of directives that the loader understands is in a later section of this manual.

go [*address*]

Start simulating the machine. If *address* is given, execution starts at that memory address. Otherwise, it continues from wherever it left off previously. This command does not complete until simulated execution stops. The return value is an information string about why execution stopped and the current state of the machine.

`help` [`command`]

When followed by a command name, it gives information on the command and how to use it. Otherwise, it gives all the available commands.

`print` [`queue`] [`window`] [`reorder`] [`unit`] [`all`] [`help`]

This command displays the contents of the machine elements specified by the options at the clock cycle when simulation last stopped. Any combination of options may be selected. The options and the result of choosing them are as follows:

`queue` display the instruction queue in a table with the following column header:

- #: entry number.
- icount: dynamic instruction count.
- address: instruction address in memory.
- opcode: the instruction opcode.
- rs1: source operand 1.
- rs2: source operand 2.
- rd: register destination.

`window` print both instruction windows: integer and floatingpoint with the following column headers:

- #: instruction entry number in the window.
- opcode: instruction opcode.
- address: instruction address (Program Counter).
- rb entry: reorder buffer entry number of the instruction
- operand1: contains either the value of the firstoperand or a reorder buffer entry from where the value will be taken when computed.

- okl: the validity field of the firstoperand: it contains 0 if the operand refers to a reorder buffer entry and 1 if the operand value is available.
- typl: type of the firstoperand (INT: integer, FPS: floating point simple, FPD: floatingpoint double IMM: immediate). The type of the operand indicates from which reorder buffer the operand value can be taken.
- operand2: second operand, just like the firstoperand, it can contain either a value or a reorder buffer entry.
- ok2: validity field of the second operand.
- typ2: type of the second operand (same as typel).
- pred: only for branch instructions, indicates what prediction has been made at fetch stage (PT: predict taken, PNT: predict not taken).

`reorder` display the integer and the foatingpoint reorder buffer, in tables with the following column headers:

- #: instruction entry number.
- icount.
- opcode.
- loc: memory location address for the result.
- unit: functional unit where the instruction executes.
- result: instruction result when produced (some instructions such as branches produce no result).
- ok: validity of the result.
- ready: indicates at what clock cycle the computation of the result will be over.
- flsh: the flush bit; it is set to 1 when the instruction follows a wrong branch prediction. In that case, when

the instruction arrives at the head of the reorder buffer, it is discarded.

`unit` show the latency, number, and current usage of the functional units.

`help` print help information on how to interpret the displayed data, for any of the above machine elements. The machine components for which information is requested must be passed as arguments to the commands. If none of the components is specified, information is provided for all of them.

`reset`

This command resets the superscalar machine, though keeping the parameters specified by the machine configuration file. The permits to run different simulations without quitting the simulator. After resetting the machine, the .s files will have to be reloaded using the **asm** command.

`stats` [`-p#`] [`-g`] [`-r`] [`-i`] [`-b`] [`-f filename`] [`-a`]

This command will dump various statistics collected during simulation. Various options are provided to take those statistics that are necessary and also to redirect the output to a file.

`-f filename` Redirects the output to a file. The file name is specified one space after f.

`-p` Set the minimum value for percentages (percentages lower than this value will not be output). The value is specified just after p. It can be an integer or a real number.

`-g` General information display.

- Number of fetched, decoded and issued instructions.
- Number of committed instructions, writes to memory locations and useless writes. A useless write is a write

to a location that could have been avoided because the writing instruction is followed in the reorder buffer by another instruction writing to the same location.

- Per cycle rates of fetch, decode, issue and commit.
- Branch information: correct and wrong prediction (if branch prediction is performed).
- Fetch stalls.
- Information on renaming
  - Renamed Operands: a table gives the distribution of the number of operands renames per clock cycle.
  - Operands searching information: a table gives the distribution of the number
    of operands searched (in the reorder buffer) per clock cycle.

`-i` Instruction process information:

- Instruction issue distribution table per clock cycle.
- Instruction delay distribution table: the distribution of the number of cycles that instructions have to wait in the central window before issue.
- Instruction commit distribution table, per clock cycle.

`-b` Occupancy rate tables of the instruction windows and the reorder buffers, throughout simulation.

`-a` Display all the above statistics. This is also the default when no options are given.

`step` [*address*]

If no *address* is given, the `step` command executes a single instruction, continuing from wherever execution previously stopped. If *address* is given, then the program counter is changed to point to *address*, and a single instruction is executed from there. In either case, the return value is an information string about the state of the machine after the single instruction has been executed.

`trace on/off filename`

This command is used to write the memory access trace to a file. This is later fed to the cache simulator to identify the performance of different cache configurations, like multi-ported caches, interleaved cache banks etc. When trace is turned on, the type of memory access, address accessed and clock cycle are written to the file. The trace can also be turned off selectively.

`quit`

Exit the simulator.

ASSEMBLY FILE FORMAT

The assembler built into **SuperSIM**, invoked using the `asm` command, accepts standard format assembly language programs. The file is expected to contain lines of the following form:

- Labels are defined by a group of non-blank characters starting with either a letter, an underscore, or a dollar sign, and followed immediately by a colon. They are associated with the next address to which code in the file will be stored. Labels can be accessed anywhere else within that file, and in files loaded after that if the label is declared as **.global** (see below).
- Comments are started with a semicolon, and continue to the end of the line.
- Constants can be entered either with or without a preceding number sign.
- Immediate values can be specified with a # preceding the value.
- The address of a variable can be accessed as a immediate value by specifying the symbol before the variable.

- The format of instructions and their operands are as shown in chapter 3.

While the assembler is processing an assembly file, the data and instructions it assembles are placed in memory based on either a text (code) or data pointer. Which pointer is used is selected not by the type of information, but by whether the most recent directive was **.data** or **.text**. The program initially loads into the text segment.

The assembler supports several directives which affect how it loads the machine's memory. These should be entered in the place where you would normally place the instruction and its arguments. The directives currently supported by **SuperSIM** are:

.align *n* Cause the next data/code loaded to be at the next higher address with the lower *n* bits zeroed (the next closest address greater than or equal to the current address that is a multiple of 2^{n-1}).

.ascii "*string1*", "*string2*", ...
Store the *strings* listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.

.asciiz "*string1*", "*string2*", ...
Similar to **.ascii**, except each string is followed by a 0 byte (like C strings).

.byte "*byte1*", "*byte2*", ...
Store the *bytes* listed on the line sequentially in memory.

.byte2 *word1*, *word2*, ...
Store the *16-bit values* listed on the line sequentially in memory.

.byte4 *word1*, *word2*, ...
Store the *32-bit values* listed on the line sequentially in memory.

.byte8 *word1*, *word2*, ...
Store the *64-bit values* listed on the line sequentially in memory.

.data [*address*]

Cause the following code and data to be stored in the data area. If an *address* was supplied, the data will be loaded starting at that address, otherwise, the last value for the data pointer will be used. If we were just reading code based on the text (code) pointer, store that address so that we can continue from there later (on a **.text** directive).

.float8 *number1*, *number2*, ...

Store the *numbers* listed on the line sequentially in memory as double precision floating point numbers.

.float4 *number1*, *number2*, ...

Store the *numbers* listed on the line sequentially in memory as single precision floating point numbers.

.global *label*

Make the *label* available for reference by code found in files loaded after this file.

.space *size*

Move the current storage pointer forward *size* bytes (to leave some empty space in memory).

.text [*address*]

Cause the following code and data to be stored in the text (code) area. If an *address* was supplied, the data will be loaded starting at that address, otherwise, the last value for the text pointer will be used. If we were just reading data based on the data pointer, store that address so that we can continue from there later (on a **.data** directive).

VARIABLES

SuperSIM uses or sets the following Tcl variables:

codeStart

If this variable exists, it indicates where to start loading code in

asm commands.

dataStart

If this variable exists, it indicates where to start loading data in **asm** commands.

insCount

SuperSIM uses this variable to keep a running count of the total number of instructions that have been simulated so far.

Appendix B

An Example Machine Description File

```
Instructions_process_per_cycle


        fetch           2
        commit          2

Memory
        size            262144
        latency         1
        accesses        4

Reorder_Buffer_sizes
        integer         20
        float           20

Instruction_Window_sizes
        integer         20
        float           20

Instruction_Queue_size  20
```

```
Branch_Buffer_size      111



Integer_Functional_Units


alu
        number          4
        latency         1


shift
        number          2
        latency         1


branch
        number          1
        latency         1



mult
        number          2
        latency         5


div
        number          2
        latency         10



Floating_Point_Units


add
        number          4
        latency         2
```

```
mult
        number          4
        latency         5

div
        number          2
        latency         10

branch
        number          1
        latency         1
```

Appendix C

Instruction Set

Opcode	Value
ADD	1
ADDF	2
AND	3
DIV	4
DIVF	5
J	6
MUL	7
MULF	8
OR	9
SLL	10
SRA	11
SRL	12
SUB	13
SUBF	14
XOR	15
TRAP	16
JEQ	17
JNE	18
JGT	19
JLT	20
JGE	21
JLE	22

Table 6: Opcode Table

All the instructions except unconditional jumps and traps have 3 operands. The format of 3 operand instructions is:

OPC DEST, SRC1, SRC2

where OPC specifies the data type, addressing modes of operand in addition to opcode which can be any of the entries in table 6. DEST, SRC1 and SRC2 specify the operands in any of the 4 addressing modes specified in table 1.

The format of Unconditional jumps is as follows:

J DEST, SRC1

where DEST specifies the address to which control has to be transferred. As our machine implements procedure calls also using jump instructions, the current PC value should be saved somewhere. The place where PC should be saved is specified by SRC1.

Traps have only one operand. They just have a trap number which is specified by DEST. As for conditional jumps, there are 3 operands. The DEST field specifies the address to which control has to shift if condition is evaluated to be true. The condition to be evaluated is the comparison of SRC1 and SRC2.

Appendix D

An Example Assembly Program

D.1 A C benchmark program

```
/* This is a sample DAXPY benchmark program which we have written.
 * It does the following operation:
 *                        Y = AX + Y
 * where X and Y are vectors (of double precision numbers) of size 'n'.
 *'A' is a double precision number which is multiplied with X[i]'s
 * and the result is added to Y[i] and the final result is stored back
 * in Y[i].
 */
#include <stdio.h>
#define         a       3.5000000000000
#define         n       100

double x[n], y[n];
main()
{
      int i;
      for(i=0; i < n; i++)
           y[i] = a*x[i] + y[i];
}
```

D.2 PERL RISC Assembly code

This section gives the assembly code generated by the C compiler [Kum97] for **PERL RISC** for the above mentioned C program. It should be clear that the C compiler compiles the system calls and library functions into traps and the simulator handles them by using the corresponding UNIX system call or the library function in the trap handler. "fp" and "sp" are 2 global variables which the simulator automatically recognizes as the frame pointer and the stack pointer which are used as base addresses as explained in chapter 3. The assembler directives **.global, float8, ...** are explained in appendix A. Many other traps which are generated by the compiler are removed as they are not relevant to this example.

First, the stack pointer is loaded with the highest address of the simulator's memory, indicated by the memSize variable. The symbol # is used to indicate an immediate value. If the address of a variable is needed, it can be obtained by preceding the variable with a @ sign. Both these are encoded as immediate values but two different symbols are used for the assembler to understand the context in which the variable occurs and for better understanding. Direct addressing is used if a variable is just specified as one of the operands. Indirect addressing is specified by enclosing the operand within parentheses as in (t1). Base addressing is specified by specifying the offset and giving the base address within parenthesis as in -16(fp). And finally, anything after a semicolon(;) on a line is ignored by the assembler and is assumed to be a comment. The format and number of operand of different types of instructions is given in appendix C. The mnemonic indicates the type of the operands. As explained in chapter 3, even though provision is available for representing the data type of each operand in the instruction, the C compiler [Kum97] doesn't use them. It encodes the data type in the mnemonic itself. For example, addb4 means all the operands are 32-bit integers.

```
.global _exit
.global t1
.global t2
.global t3
 .global t4
.global t5
```

```
.global t6
.global t7
.global t8
.global t9
.global t10
.global t11
.global t12
.global t13
.global t14

        .align 2
LC0:
        .float8 3.50000000000000000000
        .align 4
.global _main
_main:
        addb4 sp,#memSize, #0
        ;; Save the old frame pointer
        addb4 -4(sp),fp,#0
        ;; Establish new frame pointer
        addb4 fp,#0,sp
        ;; Adjust Stack Pointer
        addb4 sp,sp,#-52
        ;; Save Temporary locacations
        addb4 t3,#0,#0
        addb4 t5,#0,@_y
        addb4 t4,#0,@_x
        addf8 -28(fp),#0,LC0
L5:
        sllb4 t2,t3,#3
        addb4 t1,t5,t2
        addb4 t2,t4,t2
        addb4 t6,#0,t2
        mulf8 -20(fp),-28(fp),(t6)
        addb4 t6,#0,t1
        addf8 (t6),-20(fp),(t6)
        addb4 t3,t3,#1
        jleb4 L5,t3,#99
        ;; Restore the saved Temporary locations
        ;; Restore stack pointer
        addb4 sp,#0,fp
        ;; Restore frame pointer
        addb4 fp,-4(fp),#0
        ;; HALT
        j _exit,#0
_exit:
        trap #0
        j -8(sp),#0
_printf:
        trap #5
        j -8(sp),#0
_strcmp:
        trap #20
        j -8(sp),#0
_strncmp:
        trap #21
        j -8(sp),#0
_strcasecmp:
_stricmp:
_strcmpi:
        trap #22
        j -8(sp),#0
_strncasecmp:
_strnicmp:
        trap #23
        j -8(sp),#0
```

```
_strcpy:
        trap #24
        j -8(sp),#0
_strncpy:
        trap #25
        j -8(sp),#0
_strlen:
        trap #26
        j -8(sp),#0
t1:     .space 4
t2:     .space 4
t3:     .space 4
t4:     .space 4
t5:     .space 4
t6:     .space 4
t7:     .space 4
t8:     .space 4
t9:     .space 4
t10:    .space 4
t11:    .space 4
t12:    .space 4
t13:    .space 4
t14:    .space 4
.global _y
_y:     .space 800
.global _x
_x:     .space 800
```

Appendix E

An Example of Statistical Output

This corresponds to the statistics generated by the simulator after executing a matrix multiplication program. The machine configuration file given in appendix B was used. The size of the 3 matrices was 32x32 each.

E.1 Performance Metrics

```
Number of cycles: 428573

Instructions fetched        583567
Instructions decoded        580242     (99.43% of total fetched)

Instructions issued         578014     (99.05% of total fetched)
-> integers                 578014     (100.00% of total issued)
-> floating points               0     ( 0.00% of total issued)

Instructions committed      578010     (99.05% of total fetched)
-> integers                 578010     (100.00% of total committed)
-> floating points               0     ( 0.00% of total committed)
-> writes to memory         507153     (87.74% of total committed)
```

```
Per Cycle Rates
-> fetch        1.36 instructions/cycle
-> decode       1.35 instructions/cycle
-> issue        1.35 instructions/cycle
-> commit       1.35 instructions/cycle

Number of branches: 70857, taken 34883 (49.23%), untaken 35974 (50.77%)
Correct predictions                  69761 (98.45% of total branches)
-> taken                             34883 (100.00% of total taken branches)
-> not taken                         34878 (96.95% of not taken branches)
Wrong predictions
-> lost cycles                          72 ( 0.02% of total cycles)
-> flushed instructions               2231 ( 0.38% of total fetched)
Collisions in the BTB                    0 ( 0.00% of total branches)

Fetch Stalls: 120916 (28.21% of total cycle count)
Fetch stalls due to full buffers: 0 (0.00% of total stalls)
Decode Stalls: 122012 (28.47% of total cycle count)

 **Operands Renaming (flow/anti-dependencies)

Renamed operands:        25.96%  of total operands

 ----------------------------------------------------------------------
|     |      BOTH  UNITS       |   INTEGER UNIT       |FLOATING POINT UNIT|
| NUM |     total     |   %    |   total   |    %     | total    |   %    |
|-----|---------------|--------|-----------|----------|----------|--------|
|  0  |        159058 | 37.11% |    159058 | 37.11%   |        0 |  0.00% |
|  1  |        136297 | 31.80% |    136297 | 31.80%   |        0 |  0.00% |
|  2  |         84065 | 19.62% |     84065 | 19.62%   |        0 |  0.00% |
|  3  |         49153 | 11.47% |     49153 | 11.47%   |        0 |  0.00% |
!-----!---------------!--------!-----------!----------!----------!--------!
("NUM": number of operands renamed ;"total": total number of clock
cycles; "%": percentage of clock cycles)
```

* From the above table, it is clear that 2 operands were renamed in each of the 84065 cycles and all these were of type integer. Same inference is valid for all the entries.

**Operands Searching Information

```
 -------------------------------------------------------------------
|     |      BOTH  UNITS      |   INTEGER UNIT      |FLOATING POINT UNIT|
| NUM |    total     |   %    |   total    |   %    | total      |   %   |
|-----|--------------|--------|------------|--------|------------|-------|
|  0  |       122012 | 28.47% |    122012  | 28.47% |          0 | 0.00%|
|  1  |            1 |  0.00% |         1  |  0.00% |          0 | 0.00%|
|  3  |        32879 |  7.67% |     32879  |  7.67% |          0 | 0.00%|
|  6  |       273681 | 63.86% |    273681  | 63.86% |          0 | 0.00%|
!-----!--------------!--------!------------!--------!------------!-------!
```

("NUM": number of operands searched;"total": total number of clock cycles; "%"   : percentage of clock cycles)
* The above table gives statistics regarding searching the reorder buffers. It, for example, says that 3 operands were searched in the integer reorder buffers during each of the 32879 cycles (which is 7.67% of total cycle count).
**Instruction Issue Distribution

```
 -------------------------------------------------------------------
|     |      BOTH  UNITS      |   INTEGER UNIT      |FLOATING POINT UNIT |
| NUM |    total     |   %    |   total    |   %    | total      |   %    |
|-----|--------------|--------|------------|--------|------------|--------|
|  0  |        90339 | 21.08% |     90339  | 21.08% |          0 |  0.00% |
|  1  |       163001 | 38.03% |    163001  | 38.03% |          0 |  0.00% |
|  2  |       111709 | 26.07% |    111709  | 26.07% |          0 |  0.00% |
|  3  |        62501 | 14.58% |     62501  | 14.58% |          0 |  0.00% |
|  4  |         1023 |  0.24% |      1023  |  0.24% |          0 |  0.00% |
!-----!--------------!--------!------------!--------!------------!--------!
```

("NUM": number of instructions issued:"total": total number of clock cycles; "%"   : percentage of clock cycles

* The above statistics corresponds to the number of instructions issued in a cycle. It can be inferred that 3 instructions were issued in each of the 62051 cycles from the integer instruction window.

**Issue Delay Distribution

	BOTH UNITS		INTEGER UNIT		FLOATING POINT UNIT	
NUM	total	%	total	%	total	%
1	36971	6.40%	36971	6.40%	0	0.00%
2	345421	59.76%	345421	59.76%	0	0.00%
3	146470	25.34%	146470	25.34%	0	0.00%
4	31745	5.49%	31745	5.49%	0	0.00%
5	1023	0.18%	1023	0.18%	0	0.00%
6	15361	2.66%	15361	2.66%	0	0.00%
7	1023	0.18%	1023	0.18%	0	0.00%

("NUM": number of clock cycles;"total": total number of instructions;
"%": percentage of issued instructions)

* The Delay distribution table gives information about instructions which were placed in the instruction windows but not issued due to may be non-availability of operands, structural hazard etc.

**Instruction Commit Distribution

	BOTH UNITS		INTEGER UNIT		FLOATING POINT UNIT	
NUM	total	%	total	%	total	%
1	39060	9.11%	39060	9.11%	0	0.00%
2	269475	62.88%	269475	62.88%	0	0.00%

("NUM": number of instructions;"total": total number of clock cycles;
"%": percentage of clock cycles)

```
**Occupancy of the Integer Buffers

 -------------------------------------------------------------
|      |  INSTRUCTION WINDOW    |     REORDER BUFFER       |
| %OCC |  total cc    |   %cc   |   total cc   |   %cc     |
|------|--------------|---------|--------------|---------|
| + 0% |        87200 |  20.35% |         1035 |   0.24% |
| +10% |       106809 |  24.92% |         2200 |   0.51% |
| +20% |       217157 |  50.67% |        99514 |  23.22% |
| +30% |        17407 |   4.06% |        77987 |  18.20% |
| +40% |            0 |   0.00% |        74778 |  17.45% |
| +50% |            0 |   0.00% |        90116 |  21.03% |
| +60% |            0 |   0.00% |        32768 |   7.65% |
| +70% |            0 |   0.00% |        32768 |   7.65% |
| +80% |            0 |   0.00% |        17407 |   4.06% |
!------!--------------!---------!--------------!---------!
```

E.2 Fetch Stalls

The simulator gives various metrics which are explained in the last section. Of these, the number of fetch stalls are also output. Fetch is stalled only in the following cases:

1. During the entry point to a function. This is because, the "sp" value is changed to create a new frame and the temporaries used during this procedure call are stored in this frame. We have commands like the following:

```
;;Establish new frame pointer
addb4 fp, #0, sp
;;Adjust stack pointer
addb4 sp, sp, #-24
;;Save temporary locations
addb4 0(sp), t1, #0
addb4 4(sp), t2, #0
```

As the destination address of the third and fourth addition instructions are not known till the second add is computed, fetch is stalled after second add instruction. Note that fetch is not always stalled at the entry point of every function. It is stalled only if it is absolutely necessary. Such code appears usually at the entry point of function calls.

2. During exit from a function. Note that, procedure calls are handled by unconditional jumps. This jump actually saves the return address in the stack. At exit from function calls, we encounter instructions like:

```
;;Restore stack pointer
addb4 sp, #0, fp
;;Restore frame pointer
addb4 fp, -4(fp), #0
;;Return
j -8(sp), #0
```

   The jump instruction restores the PC value so that execution returns to the callee and fetch proceeds from that address. But we do not know what should be loaded into PC till the stack pointer is restored. Thus the fetch is stalled.

3. Due to indirection.

```
addb4 t1, t2, t1
addb4 (t1), #0, t3
addb4 t2, t2, #1
```

   In the above case also fetch is stalled after the second add instruction. This is because, the destination address of second add is known only after the first add is executed and all instruction that follow the second add stall because we do not know into which location the second instruction writes. If we allow the following instruction to proceed, it may result in a hazard which goes undetected and end up producing wrong results. This is similar to case 1 but there the culprit is based addressing and here it is indirect addressing.

```
addb4 t1, t2, #0
addb4 t3, (t1), #24
```

Note that there is a dependence due to indirection here also, but fetch is not stalled because we handle this dependence by address forwarding and this doesn't introduce any hazard with subsequent instructions. So the subsequent instructions are also fetched and issued.

As all cases of stalls are clearly defined, the compiler can do some optimizations (like instruction reordering and loop unrolling) to improve performance by reducing stalls.

Bibliography

[AAD90] D. Alpert, A. Averbuch, and O. Danieli. Performance Comparison of Load/Store and Symmetric Instruction set Architectures. In *ACM SIGARCH*, volume 18, pages 172–181, 1990.

[AD92] M. Allan and K. Diefendorff. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[BUOO94] Brad Burgers, Nasr Ullah, Peter Van Overen, and Deene Ogden. The PowerPC 603 Microprocessor. *Communications of the ACM*, 37:34–46, June 1994.

[Dig93] Digital Equipment Corporation. *Atom Reference Manual*, Dec 1993.

[EBea95] John H. Edmondson, Peter J. Bannon, and Paul I. Rubinfeld et al. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 1995.

[FMM87] M.J. Flynn, J.M. Mitchell, and J.M. Mulder. And Now a case for more Complex Instruction Sets. *IEEE Computer*, pages 71–83, Sep 1987.

[Gro90] G.F. Grohoski. Machine organization of the IBM RISC System/6000 processor. Technical report, IBM Research and Development, Jan 1990.

[Kum97] G.V. Ramana Kumar. Cache Simulation and Porting gcc to PERL-RISC. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, Jan 1997.

[LB92] L.B.Hostetler and B.Mirtich. DLXsim-A simulator for DLX. Technical report, University of California, Berkeley, Jan 1992.

[McL93] Edward McLeelan. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro*, pages 36–47, June 1993.

[Mey78] G.J. Meyers. The evaluation of expressions in a storage-to-storage architecture. *Computer Architecture News*, 7:3:20–23, Oct 1978.

[Mou93] Cecile Moura. SuperDLX-A Generic Superscalar Simulator. Master's thesis, Advanced Compilers, Architectures and Parallel Systems Group, McGill University, May 1993.

[Pat85] David A. Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28:8–21, Jan 1985.

[PH94] David A. Patterson and John L. Hennessy. *Computer Architecture—A Quantative Approach*. Morgan Kaufmann Publishers Inc., second edition, 1994.

[Sit79] Richard L. Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527–532, Jan 1979.

[Sit93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36:33–44, Feb 1993.

[Tab94] Daniel Tabak. *Advanced Microprocessors*. McGraw Hill, second edition, 1994.

[TGN95] Marc Tremblay, Dale Greenley, and Kevin Normoyle. The Design of the Microarchitecture of UltraSPARC-I. In *Proceedings of the IEEE*, volume 83, pages 1653–1671, 1995.

[Wal79] D.W. Wall. Register Windows vs Register allocation. *ACM SIGPLAN*, pages 67–78, jun 1979.

[Way96] Peter Wayner. Sun Gambles on Java Chips. *Byte*, pages 79–88, November 1996.

[web95] webmaster@sun.com. The UltraSPARC Architecture—Technology White Paper. Technical report, http://www.sun.com, 1995.